Title
Efficient Maximal Poisson-Disk Sampling

Journal
ACM Transactions on Graphics, 30

Authors
Ebeida, Mohamed S.
Patney, Anjul
Mitchell, Scott A.
et al.

Publication Date
2011

DOI
10.1145/1964921.1964944

Peer reviewed

Efficient Maximal Poisson-Disk Sampling

Mohamed S. Ebeida*
Sandia National Laboratories
Anjul Patney
University of California, Davis
Scott A. Mitchell
Sandia National Laboratories
Andrew A. Davidson
University of California, Davis
Patrick M. Knupp
Sandia National Laboratories
John D. Owens
University of California, Davis

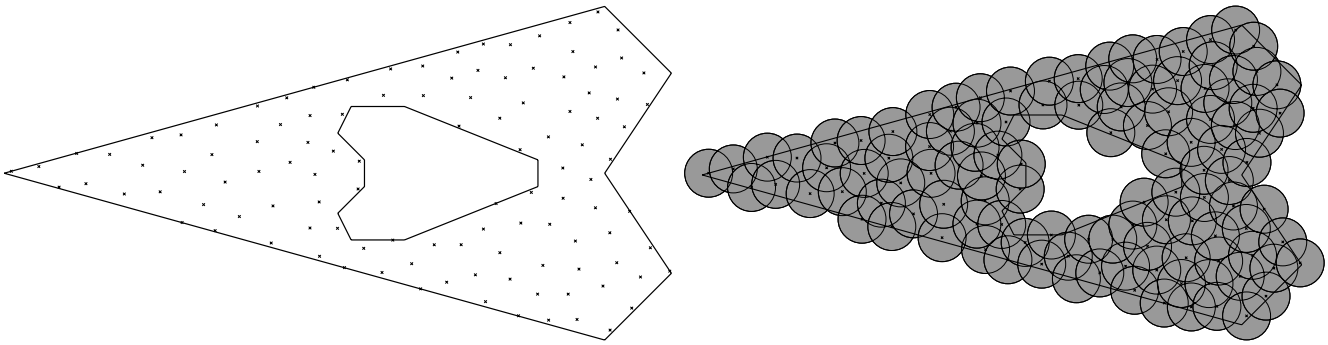


Figure 1: *A Poisson-disk sampling of a non-convex domain (left). The gray-shaded disks show the sampling is maximal (right).*

Abstract

We solve the problem of generating a uniform Poisson-disk sampling that is both **maximal** and **unbiased** over bounded non-convex domains. To our knowledge this is the first provably correct algorithm with time and space dependent only on the number of points produced. Our method has two phases, both based on classical dart-throwing. The first phase uses a background grid of square cells to rapidly create an unbiased, near-maximal covering of the domain. The second phase completes the maximal covering by calculating the connected components of the remaining uncovered voids, and by using their geometry to efficiently place unbiased samples that cover them. The second phase converges quickly, overcoming a common difficulty in dart-throwing methods. The deterministic memory is $O(n)$ and the expected running time is $O(n \log n)$, where n is the output size, the number of points in the final sample. Our serial implementation verifies that the $\log n$ dependence is minor, and nearly $O(n)$ performance for both time and memory is achieved in practice. We also present a parallel implementation on GPUs to demonstrate the parallel-friendly nature of our method, which achieves 2.4× the performance of our serial version.

CR Categories: I.3.5 [Computing Methodologies]: Computer Graphics—Computational Geometry and Object Modeling; F.2.2 [Theory of Computation]: Analysis of Algorithms and Problem Complexity—Nonnumerical Algorithms and Problems

Keywords: Poisson disk, maximal, provable convergence, linear complexity, sampling, blue noise

Links: DL PDF

*e-mail: msebeid@sandia.gov

http://doi.acm.org/10.1145/1964921.1964944

1 Introduction

Maximal Poisson-disk sampling distributions are useful in many applications. In computer graphics these distributions are desirable because the randomness avoids aliasing, and they have the blue noise property. Blue noise means the inter-sample distances follow a certain power law, with high frequencies more common. The lack of low-frequency noise produces visually pleasing results for rendering, imaging, and geometry processing [Pharr and Humphreys 2004]. The bias-free property is crucial in fracture propagation simulations. In this process, a random point cloud is required to minimize the effect of the dynamic re-meshing on the direction of the crack growth. "Regular geometries tend to form preferential directions for crack propagation." [Bolander and Saito 1998] "A randomly generated particle system, on the other hand, approximates isotropic fracture properties well." [Jirásek and Bazant 1995] A maximal distribution improves the quality bounds and performance of meshing methods such as Delaunay triangulation [Attali and Boissonnat 2004].

Poisson-disk sampling is a process that selects a random set of points, $X = \{x_i\}$, from a given domain, $\mathcal{D}$, in some K-dimensional space. The samples are at least a minimum distance apart, satisfying an empty disk criterion. In this work, we focus on the two-dimensional uniform case, where the disk radius, r, is constant regardless of location or iteration. Inserting a new point, x_i, defines a smaller domain, $\mathcal{D}_i \subset \mathcal{D}$, available for future insertions, where $\mathcal{D}_o = \mathcal{D}$. The *maximal condition* requires that the sample disks overlap, in the sense that they cover the whole domain leaving no room to insert an additional point. This property identifies the termination criterion of the associated sampling process. *Bias-free* or *unbiased* means that the likelihood of a sample being inside any subdomain is proportional to the area of the subdomain, provided the subdomain is completely outside all prior samples' disks. This is uniform sampling from the uncovered area. This definition of "unbiased" is standard in the Poisson-disk context [Gamito and Maddock 2009] and is equivalent to the Matérn second process in statistics [1960]. (And it goes by other names in other sciences.)

Bias-free: $\forall x_i \in X, \forall \Omega \subset \mathcal{D}_{i-1}$:

$$P\left(x_i \in \Omega\right) = \frac{\text{Area}(\Omega)}{\text{Area}(\mathcal{D}_{i-1})} \tag{1a}$$

Empty disk: $\forall x_i, x_j \in X, x_i \neq x_j : ||x_i - x_j|| \geq r$ (1b)

Maximal: $\forall x \in \mathcal{D}, \exists x_i \in X : ||x - x_i|| < r$ (1c)

Despite the desirability of this distribution, it has been challenging for the community to discover an efficient algorithm that satisfies all three conditions. To our knowledge, all prior methods relax the unbiased or maximal conditions, or require potentially unbounded time or space. The relaxations may be quite small in practice. The maximal condition may be resolved down to machine precision. The bias may be unnoticeable in pair-wise distance spectrum plots. But our present work appears to be the first method that provably meets all the conditions with time and space dependent only on the number of samples produced. (It appears that White et al. [2007] and Gamito and Maddock [2009] require a tree whose depth is dependent on machine precision; see below.) Our main drawbacks are the memory requirements for storing polygons and the complication of coding geometric primitives and tracking data structures. Our implementations show that any drawbacks are not overwhelming, and the method works well in practice. Our running time is competitive with the best.

For a detailed survey of Poisson sampling methods, see Lagae and Dutre [2008]. Selecting an unbiased Poisson-disk sample point is known as *dart-throwing* in computer graphics. The basic procedure is to throw a dart, random and uniformly by area. If it is already covered by a prior dart's disk, it is a "miss" and discarded; otherwise it is a "hit" and kept. The challenge is that as the number of prior darts becomes large, the uncovered area becomes smaller and its boundary becomes more complex. The classic method [Dippé and Wold 1985; Cook 1986] is to sample uniformly from the entire domain; when the fraction of new throws that are hits becomes very small, the sampling is likely close to maximal, so the algorithm terminates. This is unbiased but also not maximal in finite time. To get closer to maximal, we must take additional steps to track the uncovered domain and select new points from it.

Tile-based methods improve the performance, but relax the bias-free condition. For example, Wang tiles [Cohen et al. 2003; Lagae and Dutré 2005] require a biased Voronoi relaxation step to satisfy the empty-disk condition. Penrose tiles [Ostromoukhov et al. 2004; Ostromoukhov 2007] have a single sample per tile and require Voronoi relaxation to reduce sampling artifacts. Another class of methods improves efficiency by computing samples on the fly [Mitchell 1987; Jones 2006; Dunbar and Humphreys 2006; Bridson 2007]. However, these methods are biased and require relatively large storage. Dunbar and Humphreys [2006] proposed a linear-time advancing-front method where each new sample is picked from a region near to prior samples. Each new point has the same distance to its nearest neighbor, which violates the bias-free condition. Grid-based methods have emerged recently and are very efficient. Wei [2008] proposed a parallel sampling method that employs a sequence of multi-resolution uniform grids in the dart-throwing process. While quite practical, the phase groups are not completely bias-free, and the algorithm terminates with only a nearly-maximal distribution. Bowers et al. [2010] use a similar phase-group-decomposition method to Wei but without a hierarchy.

To get closer to a maximal distribution, White et al. [2007] uses a tree to capture the remaining uncovered area and select new samples. The memory requirements have been improved by a variation due to Gamito and Maddock [2009]. These methods are very effective in practice, and are unbiased. However, it appears to us that the authors do not claim to provide a provably maximal distribution with a data-structure size independent of numerical precision. The issue is the tree depth. The tree must be deep enough to represent the geometric gap between non-overlapping disks. In theory, this gap may be infinitely small, and thus their tree-based methods may be quite deep. In practice, they assume that darts are placed on a discrete numerical-precision grid, rather than in continuous real space. So the gap distance that needs to be represented is bounded by some function of machine precision, and the methods bound the tree depth by a predetermined constant. White et al. state, "In theory the number of active square levels could be unbounded, but in practice we only need enough levels for the precision of the number being used." Gamito and Maddock state, "A maximal-subdivision-level condition is important to prevent the algorithm from becoming locked in an infinite loop" and uncovered gaps that are too small to be captured by that level are discarded, so the maximality condition is relaxed somewhat. Figure 7 of Gamito and Maddock shows three nearly-overlapping circles with a very small uncovered region. This example is problematic for both Gamito and Maddock, and White et al., but our method handles it with no special consideration.

Gamito and Maddock, and Wei, provide algorithm descriptions for domains in arbitrary dimensions. Their papers show implementation results up to four and six dimensions respectively.

In this paper, we present an effective and provably correct algorithm to solve the two-dimensional maximal Poisson-disk sampling problem. Our algorithm inherits from Wei's algorithm [2008] many desirable properties, such as efficient parallel implementation using GPUs. Our algorithm is more complicated than Wei's, but our output is unbiased. We generate maximal distributions over non-convex domains while consuming limited resources. To our knowledge, this is the first practical algorithm that simultaneously satisfies all the requirements of a maximal Poisson-disk sampling, with complexity $E(n \log n)$ time and $O(n)$ space dependent only on the number of output points. Higher numerical precision inherently requires more memory for representing numbers and more time for arithmetic operations; however, our method requires nothing beyond that.

Our sampling process has two phases. For efficiency, we use a background grid of square cells covering the whole domain. Each cell can accommodate a single sample. In the first phase, darts are thrown into these cells. The initial darts are unlikely to overlap so the algorithm progresses quickly at first, but slows down as more darts are placed. Thus, we switch to a second phase. The first phase leaves many small empty *voids*, the part of a grid cell outside all circles. These are approximated by convex polygons. During the second phase, darts are thrown directly into the voids, with probability proportional to the relative areas of the voids, which maintains the bias-free condition. (Special care is needed because of the polygonal approximation.) The algorithm is capable of tracking the remaining voids in the domain up to round-off error using only $O(n)$ size data. A maximal distribution is achieved when the domain is completely covered, leaving no room for new points to be selected.

Our algorithm is capable of handling non-convex domains with holes, which are typical in many meshing applications. Most MPS methods focus on the unit square. To our knowledge, no prior MPS method considers non-convex domains, with or without holes.

The serial implementation of our algorithm is capable of generating one million samples from a square domain in less than 10 seconds on a modern CPU. Our parallel implementation on a GPU also produces unbiased maximal distributions and is about 2.4× faster than our CPU implementation.

In the rest of this paper, we present our algorithm in gradual steps.

In Section 2 we describe the various steps of the two-dimensional sequential algorithm, with proofs and analysis in Section 2.5. We then present our parallel implementation in Section 3. Finally, we describe application examples in Section 4 that demonstrate the efficiency of the proposed method and the quality of the output distributions.

2 Sequential Sampling

Our algorithm consists of the following steps:

1. Generate a background grid; mark interior and boundary cells.
2. Phase I. Throw darts into square cells; remove hit cells.
3. Generate polygonal approximations to the remaining voids.
4. Phase II. Throw darts into voids; update remaining areas.

While Phase I is not required for the correctness of the algorithm, it is a significant time and memory optimization. In Phase I, successful dart throwing is very fast initially, then slows down as more darts are inserted. Phase II requires computing and storing the intersection of geometric objects, and a dart is thrown into a polygon selected using a binary search. This results in an additional cost in the dart throwing procedure; however, it does not slow down as darts are inserted. (It actually speeds up!)

Our two phase-algorithm utilizes an active pool of cells to guide the dart throwing process. During Phase I, each cell in the active pool is square-shaped and can accommodate a single sample. Once a dart is thrown successfully into a randomly-selected cell, this cell is invalidated and removed from the active pool. An *invalid cell* is a cell that does not have any room for a new sample. If a thrown dart violates the empty-disk condition or lies outside the domain, the associated sample is rejected and that trial is considered a *miss*. In Phase I, we throw $A|\mathcal{C}|$ darts where $\mathcal{C}$ is the set of initial cells intersecting the interior of the domain. In addition, as a speedup, after $B|\mathcal{C}|$ throws we terminate early if there are M consecutive misses. $A = 5, B = 1/16$ and $M = 400$ works well for the serial implementation. The running-time of Phase I is linear, $O(|\mathcal{C}|)$.

The regions of cells that remain uncovered by disks after Phase I are called *voids*. They are usually small and scattered all over the domain. We loop over the remaining valid cells and calculate their intersection with prior darts to get a more accurate representation of their remaining voids (if any). For simplicity we bound the voids by chords of circles rather than the true circular arcs; this is a good approximation since the circles are large compared to the squares. A polygonal void is an outer approximation to an arc-bounded void. A polygonal void is selected based on its area relative to the sum of the areas of all the polygonal voids. A dart is thrown into the selected void, based on a uniform sampling of the polygon. As in Phase I, a dart is a miss if it is inside a prior sample, i.e. it is inside the polygonal void but outside the arc-bounded void. The chance of success is bounded below by a constant, because the area of a polygonal void is at most a constant factor greater than the arc-bounded void it approximates. Selecting a void and updating selection probabilities can be done using a tree in $O(\log \mathcal{V})$ time, where $\mathcal{V}$ is the set of remaining voids. For memory efficiency, in our implementation we use a flat array and a lazy update of the selection probabilities, updating only when many misses occur in a row. This is called a *stage*. In practice, despite the non-linear theoretical complexity, void selection is fast compared to the linear-time geometric calculations. The expected run-time of Phase II is $O|\mathcal{V} + \mathcal{V} \log \mathcal{V}|$ with a larger constant in front of the linear term.

Virtually all the methods in the literature have a hard time selecting points from small voids, doing worse as voids get smaller. We do not have this problem. Voids are selected proportional to their polygonal area. If a void is selected, the chance of successfully throwing a dart into it is actually better if its area is smaller, because the chord is a better approximation for shorter arcs.

2.1 Generation of a background grid

We generate the background grid using a variant of flood-fill (a.k.a. seed-fill, boundary-fill), a ubiquitous technique in graphics.

The input to our algorithm is a set of edges defining a polygon with holes. Boundary edges are oriented so the domain interior is to their left. Our algorithm can also accommodate point set inputs and prescribed sample points embedded in a bounded domain. Prescribed internal edges have no relevance for sampling, but are allowed because they are useful for some mesh generation algorithms.

We start the algorithm by generating a uniform grid covering a bounding box of the input domain. Cell sides are length $r/\sqrt{2}$ and cell diagonals are length r, so each cell can only accommodate a single sample. Each cell has two attributes. A *valid cell* is one that might accept a sample, and a *boundary cell* intersects one or more of the input edges. These two classifications are stored using two boolean arrays to get the best performance. Sampling a million points in a square takes three million cells, which consumes less than 1.0 MB of memory.

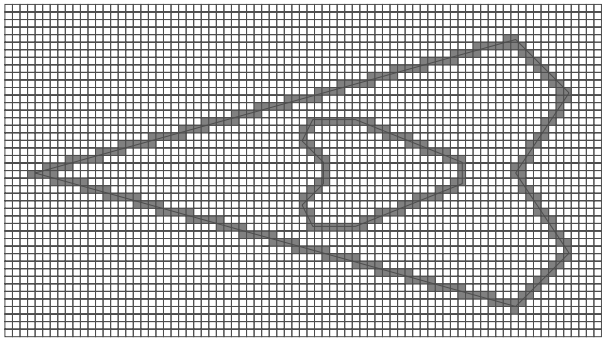
After generating the uniform grid, we identify the boundary cells. Along each edge we generate a uniform set of temporary points, and locate the hit-cell containing each point. Some cells might not have a point because an edge could graze its corner and have a short length inside the cell. These missed cells are neighbors of hit cells, and are recovered by checking the intersection of the edge with the sides of the hit cell, using only integer operations. The cells that are interior to the domain are distinguished from those exterior to the domain using a flood-fill algorithm. Exterior cells are invalid, interior and boundary cells are valid.

For the purposes of proving that we do not spend too much time on external cells when building the background grid, we assume that the number of external cells is bounded by a constant times the number of internal cells. For the purposes of proving constant complexity per void in Section 2.5.2, we assume that r is chosen small enough that a cell contains at most one domain boundary vertex and two domain boundary edges. These assumptions can be overcome using known preprocessing techniques but we skip the tedious details.

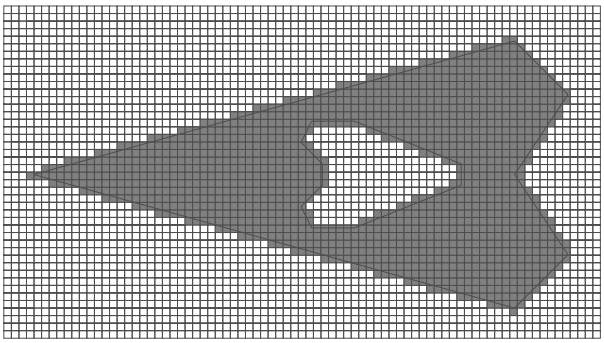
2.2 Phase I. Throwing darts into square cells

Since all the cells have the same area, $r^2/2$, the dart-throwing procedure selects a cell uniformly from the valid pool. We then draw a uniform sample from its square. If the selected sample violates the empty-disk condition or lies outside the domain boundary, that iteration is a miss. Otherwise, the dart hits a still-uncovered region. We accept the selected sample, and the cell is marked as invalid since it can no longer accept a sample. The checks are executed in constant time, since we only need to check for prior disks lying in a constant number of nearby cells and the number of boundary edges in a cell is at most two; see Section 2.5.2.

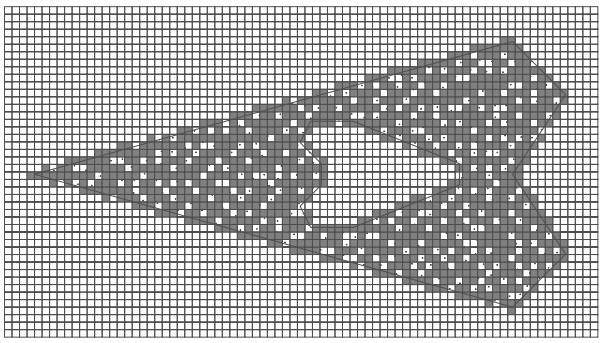
We iterate at least $|\mathcal{C}|/16$ times, where $\mathcal{C}$ is the initial set of valid cells. We continue iterating up to $5|\mathcal{C}|$ times, stopping early if 400 consecutive misses occur. At this point, the chance of successfully accepting a new sample is usually quite low, and we switch to Phase II.



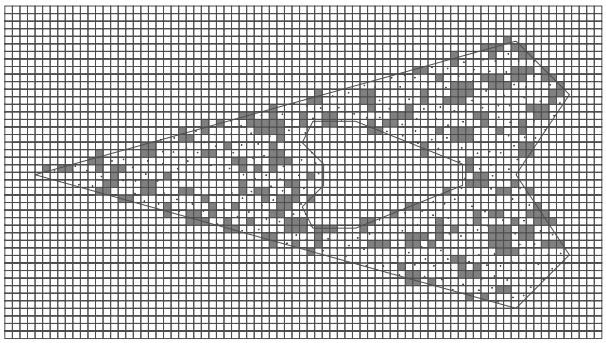
(a) Find the boundary cells (dark)



(b) Flood-fill to find valid cells (dark)



(c) Phase I darts (dots) land in many of the cells (light)



(d) Only the uncovered cells (dark) are passed to Phase II

Figure 2: *Our algorithm through Phase I.*

2.3 Polygonal approximations to arc-voids

After Phase I, for each valid cell we gather the connected components of its disk-free region inside the domain. In Phase II these components take the place of cells in Phase I. Each component is a *void*, V. It is an arc-gon, V_r, a closed 2D region bounded by straight segments and arcs of circles. We construct a polygonal outer approximation to it, V_p, representing arcs by chords. We shall prove in Section 2.5.2 that V_p is convex. A *corner* of V_p is a vertex with interior angle $< 180°$. We represent the polygon by an ordered list of its corners. The following construction algorithm is illustrated in Figure 3:

1. Initialize V_p to the cell's square, then intersect it with any boundary edges to retain just the domain interior.
2. For every disk d in a nearby cell, subtract it from V_p: for every corner c of V_p it contains,
   (a) Start from c and traverse the edges of V_p in both directions, to find the first two edges intersecting the disk's circle.
   (b) If no edges intersect the circle, then V_r is completely covered and the void is deleted.
   (c) If the chord between the points of intersection separates c from the center of d, then the disk cuts V_p into multiple connected components. Find the additional circle-edge intersections and split V_p.
   (d) Otherwise, insert two corners at the points of intersection, and remove all the intervening corners and edges from V_p, since they are covered by the disk.
   (e) Adjust the location of the new corners to be at the intersection of the arc-gon V_r and the circle.
3. If arcs of V_r intersect at any point other than a polygon corner, split the polygon into connected components, as illustrated in Figure 4.

We shall show in Section 2.5.2 that for each cell, the number of nearby disks, corners and connected components are bounded by constants. The running time of constructing each void is constant, and the running time of constructing all voids is $O(|\mathcal{V}|)$. Because of the geometric operations, the running time of this step is a significant portion of the whole.

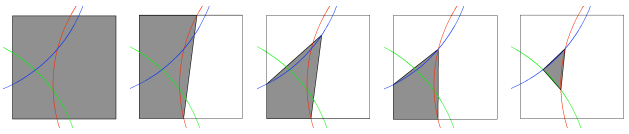


Figure 3: *Generation of the polygonal void (dark) bounded by three circles, from left to right. The polygon is initialized to the cell boundary. The red, blue, then green disks are intersected with the polygon. We get a polygon by using the chords instead of the arcs, but we update vertices at circle-polygon intersections with circle-circle intersections.*

2.4 Phase II. Throwing darts into polygonal cells

Phase II is similar to Phase I, with polygonal voids taking the place of square cells. When selecting a void we must take into account the relative areas of voids to maintain the bias-free property. After selecting a void, we choose a uniform random point inside it; see Section 2.5.3 and Graphics Gems [Turk 1993]. We use the arc-gon to determine if the selected point satisfies the empty-disk condition.

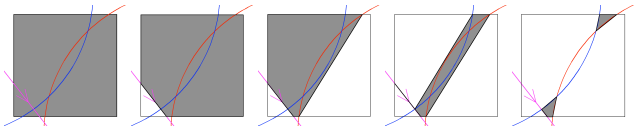


Figure 4: *Generation of two voids (dark) entrapped between two circles and a boundary edge. First the square is updated to respect the boundary edge. Next it is intersected with the circles. We detect overlapping circles containing no other polygon vertices and split the polygon in Step 3.*

If so, the process was a success, and we retain that sample. Updating the relative probabilities can be expensive, so we do that in a lazy fashion. Let $\mathcal{V}_0$ be the initial set of voids, and $\mathcal{V}_i$ the set at stage i. Similar to phase I, we throw at least $|\mathcal{V}_i|/16$ darts. We throw at most $3|\mathcal{V}_i|$ darts, quitting earlier if 100 consecutive misses occur. The expected number of hits will be a constant fraction of $|\mathcal{V}_i|$. We then update all the polygonal voids that the inserted disks overlap, using the algorithm of Section 2.3. We then compress the list of remaining voids by removing the covered ones, recomputing the relative probabilities, and continuing with pool $\mathcal{V}_{i+1}$. This continues until we have an empty pool, i.e. all voids are filled, and the distribution is maximal.

We shall prove in Section 2.5.3 that, at each throw, the probability of success is bounded below by a constant. At each stage we will fill in a constant fraction of the remaining voids. This recursion gives the total amount of work as a constant times the total amount of work in the first stage, for $|\mathcal{V}_0|$. Placing a dart and checking if it is disk-free is a constant-time operation. Updating the polygonal voids is a linear amount of work in going from stage i to $i+1$, so this is an amortized constant. The only non-constant operation is selecting which void to throw the dart in, which we next show is $O(\log \mathcal{V}_i)$. Thus the total time is $O(\mathcal{V} \log \mathcal{V})$ and the total space is $O(\mathcal{V})$.

Using a tree to keep track of the remaining uncovered regions as in Dunbar and Humphreys [2006] and Jones [2006] is a good approach if the areas are constantly updated. However, we use a simple array with half the size of a tree, and that works well for our lazy updating scheme. In practice, it appears that many cells and voids are completely covered over in the course of selecting new samples, so we do not think it is worth the computational time to constantly update voids.

Let a_i be the area of the ith polygonal void, so $p_i = a_i/A$ is the probability we should select void i. Array entry i points to the ith void, and stores $f_i = \sum_{j=1}^{i} p_i$, the sum of the probabilities of the prior voids. At each iteration we select $u \in [0,1]$ uniformly. Using binary search on the array, we find the cell with the ith percentile relative area, i.e. the i such that $f_i \geq u > f_{i-1}$. This binary search takes $O(\log \mathcal{V}_i)$ time. If the dart throw is successful, we mark the void as filled to avoid further geometric computations, but leave it in the array to avoid bias. A practical heuristic is to decide when to update the array dynamically, based on the hit rate. Updating the array after 100 consecutive misses, or when the area of the invalidated polygons exceeds $0.7A$, seems to work well.

2.5 Correctness and complexity analysis

The uninterested reader may skip ahead to Section 3. We provide some explicit values for the constants affecting the size of the data structures. Knowing the worst case allowed us to use small, fixed-size arrays in our implementation. For the constants affecting the expected running time of the algorithm, we did not try to find explicit values because they are not very useful. Instead we tuned the algorithm empirically.

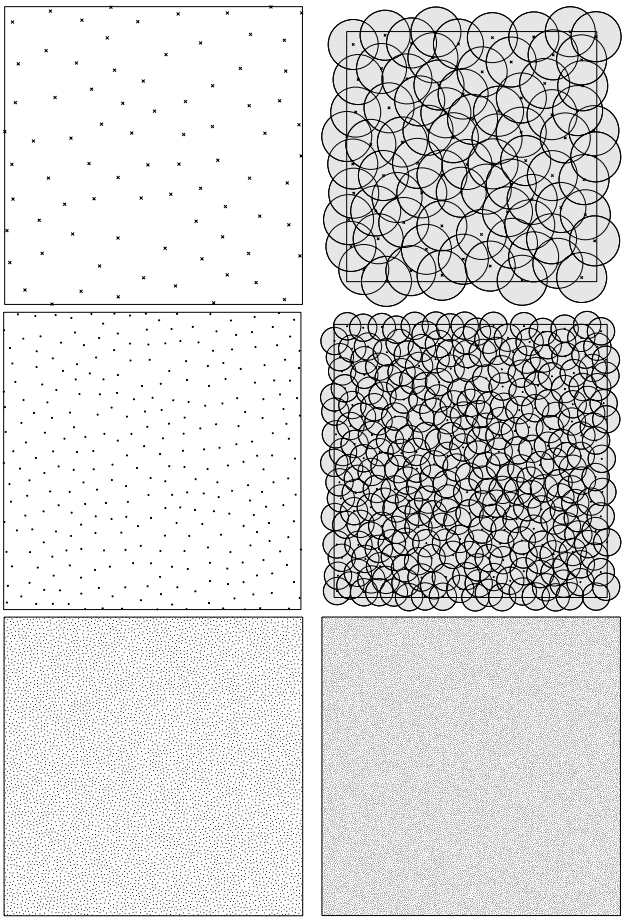


Figure 5: *Maximal Poisson samples of a unit square at four densities: $r = 0.1, 0.05, 0.01, 0.005$. For the two coarsest densities we also show their disks.*

Let n be the number of darts in the domain after the algorithm terminates. We first show that we do not have too many cells.

Theorem 1 *The total number of cells $|\mathcal{C}|$ intersecting the interior of the domain is $\Theta(n)$ in any maximal Poisson sampling.*

Proof *$|\mathcal{C}| = \Omega(n)$ because each cell contains at most one dart. For the other direction, an empty cell can only be touched by a constant number of disks, because the disks have constant radius.*

2.5.1 Bias Free

In either phase, let $\mathcal{C}_k$ for $k \in \mathcal{J}$ denote a particular cell or polygonal void. Let Ω be any domain subregion $\Omega \subset \mathcal{D}_{i-1}$. Assume for now $\Omega \subset \mathcal{C}_k$. The probability that the next point x_i will be taken from Ω is the probability of selecting $\mathcal{C}_k$ times the probability of selecting Ω within $\mathcal{C}_k$, compounded by re-throws if the dart misses the remaining domain $\mathcal{D}_{i-1}$ entirely. Let $A(\cdot)$ denote area, so

$$P(x_i \in \Omega) = \frac{A(\mathcal{C}_k)}{\sum_{\mathcal{J}} A(\mathcal{C}_j)} \frac{A(\Omega)}{A(\mathcal{C}_k)} (1 + P(\text{miss}) + P^2(\text{miss}) + \ldots).$$

Since the miss probability is $1 - A(\mathcal{D}_{i-1})/\sum A(\mathcal{C}_j)$, we have $\sum_{m=0}^{\infty} P^m(\text{miss}) = \left(\sum A(\mathcal{C}_j)\right)/A(\mathcal{D}_{i-1})$. Simplifying yields

$$P(x_i \in \Omega) = \frac{A(\Omega)}{A(\mathcal{D}_{i-1})},$$

which is precisely the requirement for the bias-free equation (1a). To extend to $\Omega \not\subseteq \mathcal{C}_k$, i.e. Ω spanning several cells or voids, simply partition subsets of Ω among cells. (For Phase I, $A(\mathcal{C}_k)/\sum A(\mathcal{C}_j) = 1/|\mathcal{J}|$.)

2.5.2 Void complexity and convexity

For the purposes of assigning a disk center to a unique square, squares are considered open on their minimal extremes, as in Figure 6. We call such squares *half-open squares*.

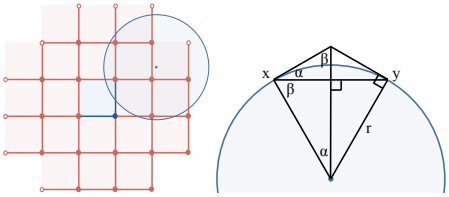


Figure 6: *Left, any r-disk intersecting the central half-open square is assigned to a unique square within this template. Right, some chord-angle identities.*

Lemma 2 ("Civilization" template) *r-disks that intersect a half-open square are assigned to one of 21 squares, up to two squares away, within a 5×5 grid of squares with the corner squares removed. See Figure 6 and the computer game Sid Meier's Civilization prior to version V.*

This shows that checking if a dart is in any disk is a constant-time operation, that any void is bounded by a constant number of disks, and that any square contains a constant number of voids. The bounds provided by Lemma 2 may be tightened by using area, angle, and distance arguments. For linearity in Phase II, it remains to show that a constant fraction of each polygonal void is outside any circle.

Lemma 3 (disks in square by area) *No more than 15 r-disks can intersect a square.*

Proof *An empty-disk system of r-disks induces a system of* non-intersecting *$r/2$-disks with the same centers but half radius. An area argument shows that only 15 of these disks can have centers close enough ($< 3r/4$) to touch the square.*

For voids we will improve the 15-disk bound to 9. Figure 8 is a construction showing that 8 disks can bound a void.

A *void* V_r is one connected component of the non-empty intersection of a square, together with the closed complement of some radius-r disks. V_r is an arc-gon. A *polygonal void* V_p is the convex hull of V_r. For convenience in the proofs, we retain *flat* vertices v of V_r as vertices of V_p when the angle of V_p at v is 180°. Note that V_r is closed and bounded, and V_p is an outer approximation to V_r. The *polygonal angle* at a vertex y of V_r will mean the interior angle between segments $\overline{xy}$ and $\overline{yz}$ where x, y, and z are consecutive vertices of V_r. $\overline{xy}$ denotes the line segment between x and y and $|xy|$ denotes the straight-line distance between x and y.

Our next series of lemmas shows that all the vertices of V_r appear as vertices of V_p, and we bound the size and shape of voids. For simplicity we assume that a square is completely interior to the domain. At the end we relax this assumption and note that the changes to the results are slight.

Lemma 4 (kites) *The angle subtended by a chord $\overline{xy}$ is twice the angle α between the circle's tangent at x and the chord. And $\alpha = \arcsin(|xy|/2r)$. See Figure 6.*

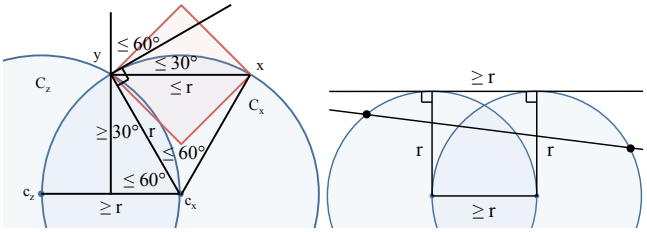


Figure 7: *Left, an upper bound on chord lengths implies an upper bound on polygonal angles. Right, only one circle bounding a void can intersect a given square side twice.*

Theorem 5 (convex corners) *The polygonal angle at a vertex of V_r is at most 180°.*

Proof *We have two cases. In the first case vertex y is at the intersection of a circle C and square edge e. At worst x and z are also on e, in which case the angle is 180°. In the second case y is at the intersection of two circles C_x and C_z; see Figure 7. The angle between the tangents of C_x and the line between the points of intersection between the two circles is at most 60°, achieved when c_x lies on C_z. The angle between the tangent of C_x at y and the chord $\overline{xy}$ is $\arcsin(|xy|/4r)$ from Lemma 4. Since the chord must lie inside the square, $|xy| \leq r$, and this angle is at most 30°. The angles between the circles-intersection line and $\overline{yz}$ are also at most 90°, so the sum of these angles is $\leq 180°$.*

Corollary 6 (naturally convex) *All vertices of V_r are on the boundary of V_p. Boundary edges of V_p are chords of circles and sub-segments of square sides.*

Corollary 7 (one arc) *A circle contributes at most one arc to V_r.*

Lemma 8 (convex centers) *The centers of circles bounding V_r must be in convex position.*

Proof *See Figure 7 left. Suppose that three circles C_x, C_y, and C_z touch V_r, but that center c_y is not in convex position with respect to centers c_x and c_z. That is, for some point p of V_r on C_y, $\angle c_z c_y p + \angle p c_y c_x > 180°$. Assume C_y intersects both of the other circles. Consider the arc of C_y between its intersection points touching V_r. (This arc is unique by Corollary 7.) Then the arc's chord is longer than r, the diagonal of the square, a contradiction. Hence the other two circles are too far away from one another to both intersect V_r. If C_y does not intersect the other two circles, then they are even farther away.*

Lemma 9 (10 arc sides) *Fewer than 10 circles bound V_r.*

Proof *We show that for r-disks bounding V_r, the distance between the centers of the two farthest-apart circles c_x and c_y must be $> 3r$, so not both can overlap a square with diagonal r. By symmetry and Lemma 8, the closest c_x and c_y can be is if all circles are arranged on a regular n-gon, with side length r. By Lemma 4 the circumscribed circle C for this n-gon has radius $R = r/(2\sin(180/n))$. Checking distances between opposite vertices for odd and even n completes the proof.*

Figure 8 shows that 8 circles is possible.

Lemma 10 (8 square sides) *The square contributes at most 8 sides to V_r, and only 4 if flat sides are ignored.*

Proof *Any circle intersecting a side exactly once (and non-tangent) contains one of the corners of the square, and so does not increase the number of subsegments of the side bounding V_r. There can only be one circle that intersects one of the four square sides twice, or is tangent to it; see Figure 7 right.*

Putting the prior lemmas together we have the following theorem.

Theorem 11 (few arc-gon sides) *The number of sides of V_r is at most 17, and at most 13 if flat sides are removed. Figure 8 realizes a void with 16 sides, and Figure 8 shows a void with 12 non-flat sides.*

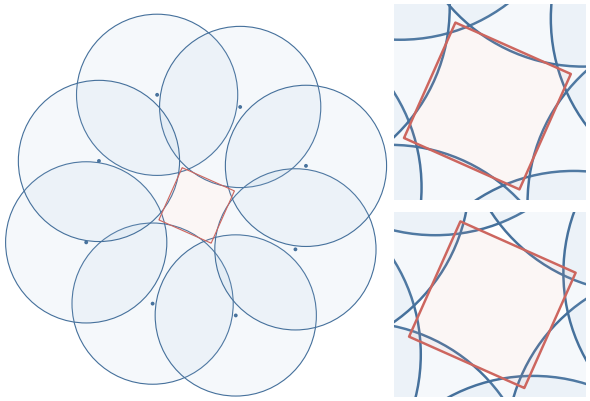


Figure 8: *A void with 16 sides (left) and a closeup (top-right). A slight tweak yields a void with 12 nonflat sides (bottom-right).*

The preceding considers only squares that did not contain the boundary of the input domain, but most of the proofs only rely on squares being contained in a circle of radius $r/2$. For boundary squares, we note that they may have at most two edges of the input domain (sharing a common vertex), and their segments on the boundary of V_r are of length $< r$, so that the number of sides increases by at most 2.

2.5.3 Phase II geometry and complexity

Area ratio of polygonal void to arc-gon void. We now consider the ratio of the area of V_r to V_p, since that determines the expected number of dart-misses in Phase II.

Theorem 12 *$A(V_r)/A(V_p)$ is at least a constant.*

Proof *Consider the circles bounding a void, including circles intersecting a square side twice. Consider the weighted Voronoi region of the circles [Edelsbrunner and Shah 1992]. Assume for now that the remainder region is bounded entirely by r-circles, and truncate the Voronoi cells at the polygonal void V_p.*

For any circle C, its Voronoi cell will contain the circle chord on the polygon boundary χ, the arc-boundary s, and a part of the interior of V_r. The reasons are as follows. Let VC be the circle's truncated Voronoi cell, and VS_{ext} its partition outside C, and VS_{int} its part inside C. Recall from Lemma 8 the circle centers are in convex position and can be considered in order around the boundary of the void. Since only the consecutive circles around the void may overlap with C (else the void would not be connected), the chord is not inside any other circle, so it is in VC. Also, the Voronoi line of equal distances between C and a non-consecutive circle lies strictly outside C. Since by Lemma 9 there are at most a constant number of circles (< 10), there are a constant number of straight sides bounding VC. All of these bounding sides lie outside VS_{ext} as well. At worst these sides approach tangency with C, and form a 9-sided polygonal outer approximation to the arc. Since the arc s has constant curvature, the area of VS_{ext} is at least a constant fraction of VS_{int}. We do not work out the exact constant because this bound is not very tight; for example, many fewer than 9 circles can be packed close enough to be nearly tangent with C.

Now relax the assumption that the remainder region is bounded entirely by circles. Treat the lines supporting the square sides or domain boundary as infinite-radius circles centered at infinity, and all the arguments of the prior paragraph still hold. The area ratio bound constant can be reproduced by assigning the Voronoi regions of the infinite-radius circles to the closest r-circle, since for the infinite-radius circles the arc-gon and polygon are identical.

Constant fraction progress per stage. Theorem 12 proves that the first dart thrown in a stage i has a constant probability of being a hit. However, there is a technical difficulty for subsequent darts. The first disk may cover other polygonal voids, perhaps completely. We update the polygons lazily, so those voids reduce the probability of a successful hit. This is resolved by recalling that any inserted disk can affect only a constant number of other voids. After $c_1|\mathcal{V}_i|$ hits, $c_2|\mathcal{V}_i|$ voids remain unchanged, so the probability of a dart being a success is at least c_2 times what it was at the start of stage i. Here c_1 is something smaller than $1/60$, and $c_2 = (1 - 60c_1)$. The 60 arises from Lemma 3 where each placed disk intersects at most 15 other cells, and by Theorem 21 each cell has at most 4 voids. Thus a lower bound on the expected number of hits in stage i is c_2 times the constant from Theorem 12 times the number of throws $c_1|\mathcal{V}_i|$: the point is this is $O(|\mathcal{V}_i|)$. In practice, many more voids are filled than these constants suggest, but the above is sufficient to prove the following theorem.

Theorem 13 *In each Phase II stage i, a constant fraction of the $\mathcal{V}_i$ voids are filled with darts.*

Success rate in practice. The constants given from the proofs of Theorem 12 and Theorem 13 are not tight. These constants do not affect any data structures in our algorithm, only the miss rate. Their importance is in the tuning of the algorithm parameters for when to move to the next stage. Our implementation shows the area ratio as in Theorem 12 is usually large, about 1. It is about 0.93 in the beginning stages of Phase II, and reaches about 0.999 in the last stage. It tends to increase but is not monotonic. See Figure 9. Also, the fraction of voids filled per stage is much better than the weak constants from Theorem 13 might suggest, as many fewer than 60 voids are touched by a new disk on average. Also the fraction goes up as the domain gets filled, as the voids become more isolated. See Figure 10.

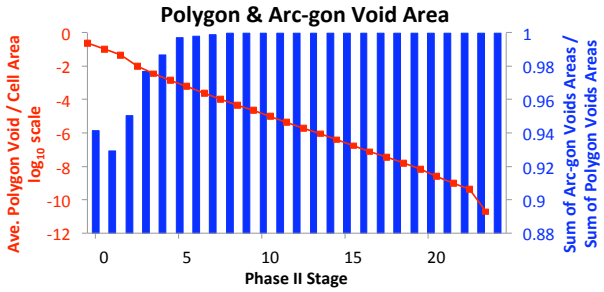


Figure 9: *The ratio of the polygon to arc-gon area is always large, almost 1. The average ratio tends to increase by stage as voids become smaller in area.*

Running time. We now consider the running time $R(|\mathcal{V}_i|)$ of Phase II for stage i and all subsequent stages. We showed in Section 2.4 that $R(|\mathcal{V}_i|) = |\mathcal{V}_i| \log |\mathcal{V}_i| + R(|\mathcal{V}_{i+1}|)$. Since $|\mathcal{V}_{i+1}| < c|\mathcal{V}_i|$ for some constant $c < 1$, we have $R(|\mathcal{V}_0|) < \sum_{i=0}^{\infty} c^i |\mathcal{V}_0| \log \left(c^i |\mathcal{V}_0|\right)$ $< \left(|\mathcal{V}_0| \log |\mathcal{V}_0|\right) \sum_{i=0}^{\infty} c^i = \frac{1}{1-c} |\mathcal{V}_0| \log |\mathcal{V}_0|$. Combining this with Theorem 1 we have

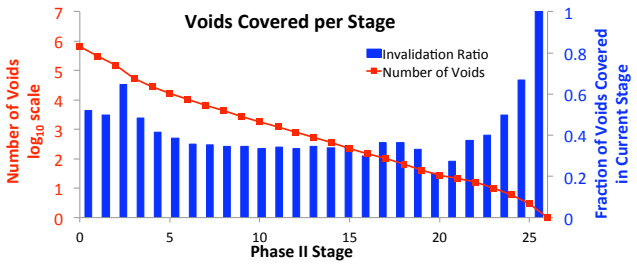


Figure 10: *The fraction of voids filled in each stage is usually large.*

Theorem 14 *Phase II running time is $O(n \log n)$.*

Uniform sampling from polygons. Recall that we needed to sample uniformly from the polygonal voids. We recap the method we use, adapted from Graphics Gems [Turk 1993]. We triangulate the polygon. Since it is convex, we could simply pick any one vertex and connect it to all the others. But for numerical reasons, it is better to keep angles away from 180°. We instead introduce a new vertex inside the polygon, located at the average of all the other vertices, and connect it to all the polygon vertices. We select one of these triangles with probability proportional to its area relative to the polygon area. Within $\triangle abc$, we sample uniformly from it by picking uniformly from a right triangle and linearly transforming to our triangle. Pick $u, v \in [0, 1]$ uniformly. This picks a point from the square; if $u + v > 1$, then reflect back into the triangle by assigning $u := 1 - u$ and $v := 1 - v$. The resulting sample point $p = u\overrightarrow{ab} + v\overrightarrow{ac}$ is uniform from $\triangle abc$.

2.5.4 Constant number of voids per cell

Since at most a constant number of circles intersect a square, combinatorics implies the number of voids in a given cell is constant. Improving it is interesting and allows some implementation efficiencies, but is not essential. These observations also hint at the observed separation distances between voids as the stage increases.

We first consider voids bounded entirely by circle arcs, then we shall see that allowing voids to be bounded by the sides of the square increases the number of voids per square by at most one. Two voids are *adjacent* if they are bounded by the same pair of circles C_x and C_y. The vertices of circle intersection are labeled vertex a_{xy} and b_{xy}, where b lies inside the reference void. Overlapping circles bounding a void are *consecutive*. We first consider three-sided remainder regions, and label their features as in Figure 11. *Consecutive adjacent voids* are two voids in the same cell adjacent to the third reference void through its adjacent consecutive circles C_xC_y and C_xC_z: e.g. two voids inside regions A_{xy} and A_{xz} in Figure 11, provided they and some part of V are in the same square cell.

Lemma 15 *For consecutive adjacent voids V_{xy} and V_{xz} to V, their closest pair of points are no closer than the circle intersection points, a_{xy} and a_{xz}.*

Proof *See Figure 11. Pick some point q of V in the cell. Since q is in a void, its distance to c_x is at least r. In particular, all of V_{xy} and V_{xz} must be on the same side as q of the line through c_x perpendicular to $\overline{qc_x}$. All of V_{xy} and V_{xz} must be within r of q, inside the red circle. The closest pair of points inside the red circle are a_{xy} and a_{xz}.*

Consider moving one of three pair-wise overlapping circles. We observe the following inverse relationships about the distances between pairs of circle centers and pairs of void vertices.

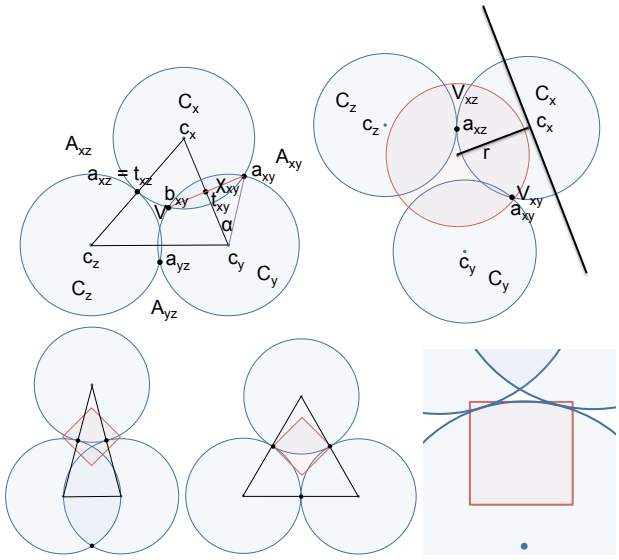


Figure 11: *Top left, labeling of a three-sided void. If C_x and C_z are tangent, then $t_{xz} = a_{xz} = b_{xz}$ coincide. Top right, the closest points of consecutive adjacent voids sharing circle C_x are not closer than $|a_{xy}a_{xz}|$. Bottom left, the 3-sided void with smallest distance between an adjacent pair of voids. Bottom center, the 3-sided void with the smallest distance between the second-closest pair of adjacent voids. Bottom left, four voids in a square.*

Lemma 16 (circle void distances) *If a_{xy}, a_{yz} and a_{xz} are in the same cell, then*

$$|c_yc_z| \uparrow \implies |a_{xy}a_{xz}| \uparrow, |a_{xy}a_{yz}| \downarrow, |a_{xz}a_{yz}| \downarrow$$
$$|c_xc_y| \uparrow \implies |a_{xz}a_{yz}| \uparrow, |a_{xz}a_{xy}| \downarrow, |a_{yz}a_{xy}| \downarrow$$
$$|c_xc_z| \uparrow \implies |a_{xy}a_{yz}| \uparrow, |a_{xy}a_{xz}| \downarrow, |a_{yz}a_{xz}| \downarrow$$

We omit the proof because of space limitations. The proof is based on Lemma 4, Lemma 8, and bounding distances by r. Lemma 16 can now be used to prove the extreme cases in Figure 11 and the following lemma.

Lemma 17 *For three-sided voids, the distance between consecutive adjacent voids is at least $r/2$. For other voids, the distance between consecutive adjacent voids is at least r.*

Corollary 18 *For a void with four or more sides, only two adjacent voids can be in the same cell as the void, and only one strictly inside.*

Proof *The square diagonal is r, so for three consecutive adjacent voids, only one pair of consecutive adjacent voids can be placed inside the square. For non-consecutive voids V_{12} and V_{34} adjacent to the void V, consider the two pairs of circles separating them from V, C_1, C_2 and C_3, C_4. If the length $\overline{c_1c_2}$ and $\overline{c_3c_4}$ are both $2r$, and $\overline{c_1c_4}$ and $\overline{c_2c_3}$ are both r, then we have a parallelogram and the distance between the circle-center midpoints t_{12} and t_{34} is r. This is a slight variation of Figure 12 where the parallelogram diagonals must be strictly greater than $2r$.*

We next argue that this parallelogram is the worst case. If $|c_1c_3|$ or $|c_2c_4|$ is greater than r, this merely makes V_{12} and V_{34} further apart. A variation of Lemma 16 and Lemma 17 shows that this is the worst case. If t_{12} is close enough to t_{34} to be of interest, then since the lengths $|c_1c_2|$ and $|c_3c_4|$ are bounded between r and $2r$,

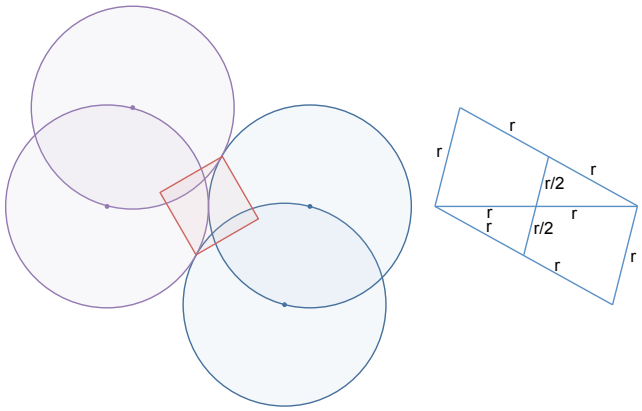


Figure 12: *The configuration minimizing the distance to an adjacent void for two voids simultaneously. The circle centers form a parallelogram, with vertices from four voids along the diagonal of a square. Only three circle-bounded voids can fit strictly inside a cell.*

the angle between lines $\overline{c_1c_2}$ and $\overline{c_3c_4}$ is small, meaning the lines are close enough to parallel, so that decreasing $|c_3c_4|$ makes a_{12} farther from a_{34} rather than closer.

Theorem 19 *For a three-sided void, only two adjacent voids can be strictly inside the same cell.*

Proof *We consider the configuration minimizing the second-closest pair of adjacent regions, and show that this distance is large. WLOG $|a_{xy}a_{xz}| \leq |a_{xz}a_{yz}| \leq |a_{yz}a_{xy}|$. We seek to show the first inequality is equality by contradiction. Suppose the first inequality is strict. Then Lemma 16 shows that increasing $|c_yc_z|$ increases $|a_{xz}a_{yz}|$ and decreases $|a_{xz}a_{yz}|$ and $|a_{xy}a_{yz}|$, a contradiction to the configuration being optimal. So $|a_{xy}a_{xz}| = |a_{xz}a_{yz}| \leq |a_{yz}a_{xy}|$. Again Lemma 16 shows that increasing $|c_xc_z|$ will decrease both $|a_{xz}a_{xy}|$ and $|a_{xz}a_{yz}|$, so the optimal configuration has maximal $|c_xc_z|$, or $|c_xc_z| = 2r$. Hence all the disks are at distance $2r$ from one another, and each of the three adjacent cell vertex pairs are distance r apart; see Figure 11. Since the square only has diagonal length r, not all three of these vertices can fit.*

Using Lemma 17 for three-sided voids and Corollary 18 for four-or-more sided voids proves the following.

Theorem 20 *A cell can contain at most three circle-bounded voids.*

Theorem 21 *A cell can contain at most four voids.*

Proof *In the prior theorems, at most two overlapping circle pairs can separate the square into multiple connected components. The main idea of this theorem's proof is that a single circle can also separate one connected component of a square into two, by overlapping with a square side. The square is too small compared to the circle radius for there to be two such circles.*

3 Parallel Algorithm

While our serial approach is fairly fast, a parallel implementation provides the opportunity to speed up sample generation, and to deploy our algorithm in GPU-based environments, where parallelism is critical for performance. Fortunately, the approach is straightforward to parallelize. In this section we describe the parallel-specific details and our GPU implementation. We use the same two phases as in our serial algorithm.

Phase I Our first phase, the dart-throwing phase (Section 2.2), requires special care to eliminate bias. Doing this in parallel with no bias has many pitfalls; the use of phase groups by Wei [2008], for instance, makes sample choices biased towards the simultaneous acceptance of cells in the same phase group, and hence picks candidates non-uniformly from the input domain.

Our algorithm needs to allow samples to have an equal probability of being selected everywhere. Thus we (must) allow samples to be chosen in parallel such that conflicts between them are possible, but we then eliminate the conflicts in an unbiased fashion. In our implementation, we achieve this through two sample buffers, *candidates* and *final*, to store the locations of points. We also store a state per cell, where each cell is either *empty*, *test*, *accepted*, or *done*. All cells are initialized to *empty*. We then implement our algorithm as three successive kernels, and repeatedly iterate over these three steps until we move to Phase 2.

First, in parallel, we generate candidates by picking a random point [Tzeng and Wei 2008] while checking for *done* points in its vicinity. We check against all nearby cells that might contain a conflicting point. We check a 5×5 neighborhood centered around the point's cell for other cells that are already marked as *done*. Our cell is marked as *test* only if no conflicts are found in this neighborhood; it then becomes a candidate for addition to the distribution.

Second, we test for conflicts again, this time for each *test* sample against other *test* candidates. We look at all *test* cells in parallel, looking for cells in the neighborhood with either *test* or *accepted* states. If we find no conflicts with other candidates, we mark the cell as *accepted*.

Third, again in parallel, we promote all *accepted* cells to *done* and copy their positions from the *candidate* buffer to the *final* buffer, and reset all remaining *test* (conflicted) cells to *empty*.

In summary, on each iteration, we first test against points already in the final distribution; next test against candidate points generated in this iteration; and finally add successful candidates to the final distribution.

Both the serial and parallel algorithms share a similar criterion for switching to Phase II. However, while the serial algorithm simply casts one candidate point per iteration until it reaches this criterion, the parallel algorithm must decide how many points to cast on each iteration. Too few and the algorithm will take too many iterations to converge; too many and many candidates will knock each other out. For n grid cells, we have found a good balance by casting $n/5$ points on each iteration.

Phase II Phase II has a similar structure to Phase I, with four steps: constructing void polygons, generating candidates in the void polygons, testing conflicts, and updating states. For this Phase, we also add an additional cell state, *rejected*, for cells that are completely covered by nearby disks and hence can never contain a point. We iterate over these steps and terminate when no void polygons remain.

We construct void polygons by visiting all *empty* cells in parallel, looping over their local neighborhood to find all points whose discs intersect with this cell. The procedure is similar to the serial algorithm: begin with a polygon that is the size of the cell, then successively subtract covered areas. We split polygons if necessary. If an unsplit polygon has zero area, the cell is labeled *rejected* and no polygons are generated.

Since each polygon is convex, we convert it into triangles and add it to a triangle buffer. We also store its area in another buffer. The

latter area buffer then undergoes a parallel exclusive prefix-sum operation [Sengupta et al. 2007] to obtain the cumulative area for each triangle. At the end of this step we also know the total area of all void polygons.

In the second step, we generate test points in some of the void triangles. This is similar to generating candidates in Phase I, but now we must pick triangles with a probability proportional to their areas. We run several threads in parallel for this purpose. Each thread picks a random number between 0 and 1, and performs a binary search over the area and cumulative area buffers to identify the triangle whose area fraction covers this number. The thread then tries to insert a *test* sample uniformly in this triangle, but backs off if another thread has already picked the associated grid cell. Each thread performs several tries for different random numbers before giving up.

The third and fourth steps are identical to Phase I's second and third steps, checking each *test* point against nearby ones. We iterate Phase II until all polygons are consumed. Very small polygons last many iterations because their areas are a small fraction of the overall polygon area, but as larger polygons are removed, their area fractions rise and they will all be visited before completion.

4 Implementation Performance

In this section we show the performance of our serial and parallel implementations of the algorithm. The serial implementation is tested using a Intel Core i7 CPU M620 with 4 GB of DRAM running a 64-bit Windows 7 operating system. We start by showing the significance of Phase II in achieving a maximal distribution with reasonable performance. In our algorithm, Phase I mimics an improved version of the classical dart throwing algorithm. This provides a useful method to distribute an initial set of bias-free random points covering most of the domain. However, the capability of this phase to insert new points deteriorates as more points are inserted. Hence such an approach cannot by itself achieve maximal distributions. This fact is demonstrated in Figure 13, where 70,000 darts were thrown into a unit square domain during Phase I. At the beginning of Phase I, the percentage of successful darts is close to 100%, and as more points are inserted, this percentage decreases significantly. After Phase I ends, only 5940 points were distributed in the domain, consuming about 30 ms and covering about 98% of the total area of the domain. Phase II was able to reach a maximal distribution by inserting an additional 2175 points in the remaining 2% of the area. Limiting the number of the darts thrown in Phase I in a typical implementation of our algorithm achieves a similar result in less than 10 ms.

We compare our times for maximal sampling to White's [2007] and Gamito and Mattock's [2009] times for nearly-maximal sampling, with truncated tree depth. Our sequential implementation samples 100k points/second, on par with White's low-memory algorithm, and our method does not slow down as much when the number of points increases. Gamito reports 100k points in 1.9 seconds.

Figure 14 shows the memory consumption over the two phases of the algorithm. The domain was a unit square. We generated 8,269,890 points. The memory required was 1.970 GB, of which about 660 MB was for the output point cloud. The peak memory was when we built the polygonal voids at the beginning of Phase II. This suggests that memory could be reduced, at the cost of slower performance, by forcing Phase I to throw more darts. The saw-tooth in the figure arises because the memory jumps at the beginning of a stage when we compute the void polygons, is mostly flat during sampling, then drops at the end of a stage when we discard voids. Voids are recomputed from scratch at the beginning of each stage.

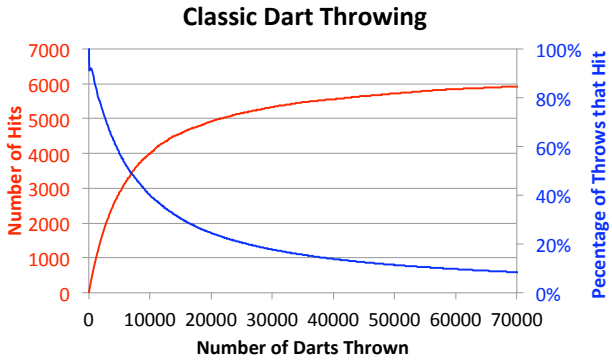


Figure 13: *The capability of classic dart-throwing to insert a new point deteriorates as the number of prior darts increases. At 70,000 darts thrown, 90% are rejected. 80% of the accepted darts were thrown during the first 20,000 throws.*

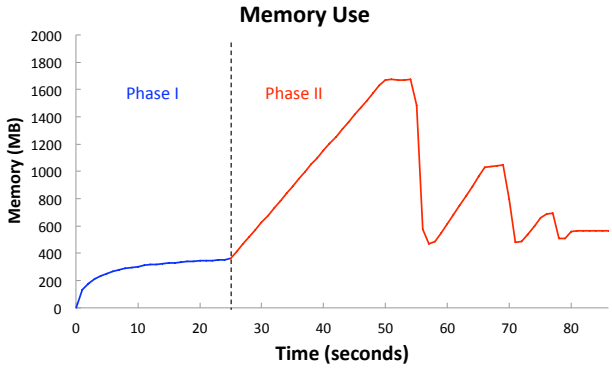


Figure 14: *Serial memory use while sampling 8 million points in a square. The memory ramps up at the start of each Phase II stage when the geometry of polygonal voids is calculated. Later stages have fewer voids. In this example three teeth corresponding to the first three Phase II stages are visible. The roughly-flat region after the third tooth is actually comprised of about 10 stages that consume little memory. This figure also illustrates that geometric void calculations are a large part of the running time.*

This avoids the cost of updating voids, many of which no longer exist.

Compared to White [2007], we consume more memory. Polygonal voids appear more expensive to represent than truncated-depth quadtrees. Asymptotically, Gamito and Mattock [2009] require $O(n \log n)$ space vs. our $O(n)$.

Figure 15 shows the runtime of the algorithm. Note the binary search in Phase II has a negligible effect on performance in practice. The memory consumption in Phase II is proportional to the number of the remaining voids after Phase I. As illustrated in this figure, the relation between the number of voids and the number of points in the final distribution is almost linear. Moreover, more than 70% of the points are inserted during Phase I. Note that these results may vary according to the input geometry as well as the termination criterion of Phase I. Here we are using a unit square problem with no holes, where Phase I terminates after 400 successive misses.

GPU Implementation. Our GPU implementation was built on the NVIDIA CUDA platform and runs on an NVIDIA GeForce GTX 460 with 1 GB of on-chip memory. The algorithms used

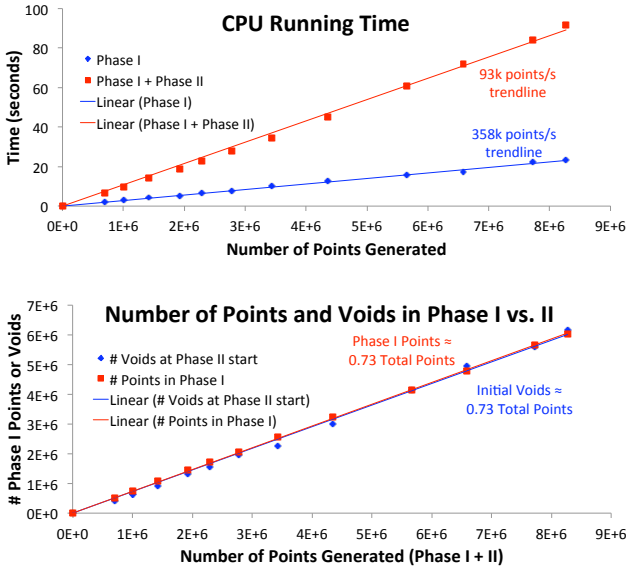


Figure 15: *CPU performance while sampling a square at different densities. The upper figure shows near-linear runtime. The lower figure shows that Phase I inserts about a constant fraction of the total number of points, and Phase I creates nearly the same number of points as voids.*

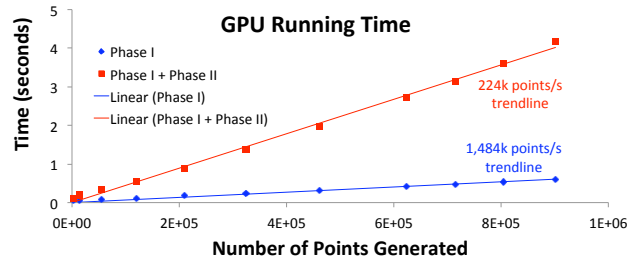
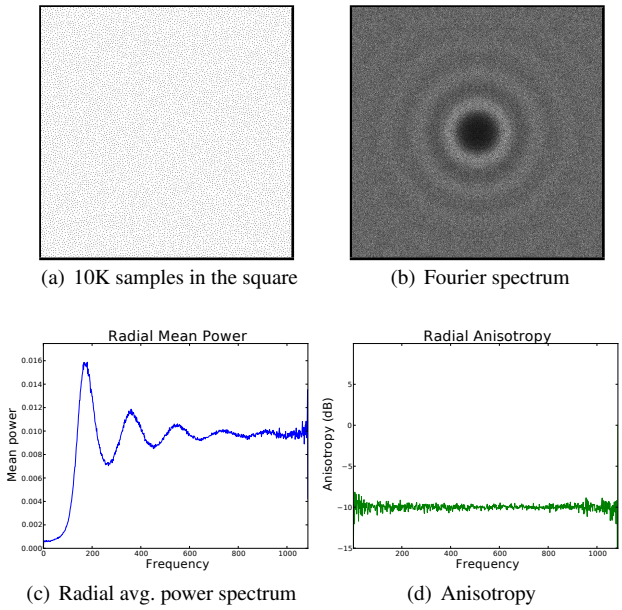


Figure 16: *GPU performance. We obtain roughly linear performance in the number of points, and a 2.4× speedup over our CPU implementation. Peak performance is about 240K samples/second. Due to memory constraints we were unable to generate more than 2 million points.*

for Phases I and II are described in Section 3. Figure 16 shows the obtained running times of our implementation for a varying numbers of final samples. We observe a linear growth in the time taken for the two phases, which is consistent with the algorithm and the CPU implementation. We also observe a 2.4× speedup over our CPU implementation, reaching a peak generation rate of about 240k points per second. Although this is not a huge speedup, our goal is to demonstrate a proof-of-concept approach that attains reasonable performance. The obtained speedup demonstrates the parallel-friendly nature of our approach.

Output Quality. Figure 17(b) shows the frequency spectrum of a set of 10K samples, and figures 17(c) and 17(d) show the radial mean and anisotropy of the spectrum across 10 trials. These measures match the expected unbiased behavior seen in previous literature [Lagae and Dutré 2008; Wei 2008], which clearly indicates the absence of any noticeable bias in our approach.



(a) 10K samples in the square (b) Fourier spectrum

(c) Radial avg. power spectrum (d) Anisotropy

Figure 17: *A parallel sampling of the square (a) and its pair-wise distance spectrum (b). There is no visible bias: the spectrum is visually identical to the results of pure dart-throwing. The radial power spectrum (c) and anisotropy (d) over ten trials also show no bias.*

5 Conclusions

We present an efficient algorithm for maximal Poisson-disk sampling in two-dimensions. The algorithm is notable because it simultaneously achieves the following: (1) the final result is provably maximal, (2) the sampling is unbiased (in the sense of dart-throwing), (3) it is $O(n \log n)$ in expected time, and (4) it is $O(n)$ in deterministic memory required. The algorithm is not limited to convex domains, and has been efficiently implemented in both sequential and parallel forms.

Empirical investigations of the sequential algorithm suggests that the order $O(n \log n)$ dependence is weak and that practical performance is very close to order n. We are able to generate 100,000 samples/second on the unit square on a modern laptop.

In the parallel implementation, we must choose samples in parallel. Conflicts that occur must be eliminated in an unbiased fashion.

We have begun the implementation of a 3D maximal Poisson-disk sampling algorithm following the ideas in this work. We are optimistic that similar results will hold for the 3D case. We are also optimistic that the algorithm may be extended to non-uniform disk radii.

Acknowledgements

We thank Joseph E. Bishop, Mike Stone, Laura Swiler, David G. Robinson, and Vitus Leung (all Sandia) for useful discussions. We thank Thouis "Ray" Jones for discussing weighted sampling algorithms with us. The Sandia authors thank the U.S. Department of Energy and Sandia's Computer Science Research Institute for supporting this work. This work was also funded in part by DOE's Office of Advanced Scientific Computing Research, SC-21, SciDAC-e. The UC Davis authors thank the SciDAC Institute for Ultrascale Visualization, the National Science Foundation (grant

CCF-1017399), an NVIDIA Research Fellowship, and the Intel Science and Technology Center for Visual Computing for supporting this work.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

ATTALI, D., AND BOISSONNAT, J.-D. 2004. A linear bound on the complexity of the Delaunay triangulation of points on polyhedral surfaces. *Discrete & Computational Geometry 31*, 3 (Feb.), 369–384.

BOLANDER, J. E., AND SAITO, S. 1998. Fracture analyses using spring networks with random geometry. *Engineering Fracture Mechanics 61*, 5-6, 569 – 591.

BOWERS, J., WANG, R., WEI, L.-Y., AND MALETZ, D. 2010. Parallel Poisson disk sampling with spectrum analysis on surfaces. *ACM Transactions on Graphics 29* (Dec.), 166:1–166:10.

BRIDSON, R. 2007. Fast Poisson disk sampling in arbitrary dimensions. In *ACM SIGGRAPH 2007 Sketches*, 22.

COHEN, M. F., SHADE, J., HILLER, S., AND DEUSSEN, O. 2003. Wang tiles for image and texture generation. *ACM Transactions on Graphics 22*, 3 (July), 287–294.

COOK, R. L. 1986. Stochastic sampling in computer graphics. *ACM Transactions on Graphics 5*, 1 (Jan.), 51–72.

DIPPÉ, M. A. Z., AND WOLD, E. H. 1985. Antialiasing through stochastic sampling. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, 69–78.

DUNBAR, D., AND HUMPHREYS, G. 2006. A spatial data structure for fast Poisson-disk sample generation. *ACM Transactions on Graphics 25*, 3 (July), 503–508.

EDELSBRUNNER, H., AND SHAH, N. R. 1992. Incremental topological flipping works for regular triangulations. In *Proceedings of the Eighth Annual Symposium on Computational Geometry*, 43–52.

GAMITO, M. N., AND MADDOCK, S. C. 2009. Accurate multidimensional Poisson-disk sampling. *ACM Transactions on Graphics 29*, 1 (Dec.), 8:1–8:19.

JIRÁSEK, M., AND BAZANT, Z. P. 1995. Particle model for quasibrittle fracture and its application to sea ice. *Journal of Engineering Mechanics 121*, 1016–1025.

JONES, T. R. 2006. Efficient generation of Poisson-disk sampling patterns. *journal of graphics tools 11*, 2, 27–36.

LAGAE, A., AND DUTRÉ, P. 2005. A procedural object distribution function. *ACM Transactions on Graphics 24*, 4 (Oct.), 1442–1461.

LAGAE, A., AND DUTRÉ, P. 2008. A comparison of methods for generating Poisson disk distributions. *Computer Graphics Forum 27*, 1 (Mar.), 114–129.

MATÉRN, B. 1960. Spatial variation. *Meddelanden från Statens Skogsforskningsinstitut 49*, 1–140.

MITCHELL, D. P. 1987. Generating antialiased images at low sampling densities. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, 65–72.

OSTROMOUKHOV, V., DONOHUE, C., AND JODOIN, P.-M. 2004. Fast hierarchical importance sampling with blue noise properties. *ACM Transactions on Graphics 23*, 3 (Aug.), 488–495.

OSTROMOUKHOV, V. 2007. Sampling with polyominoes. *ACM Transactions on Graphics 26*, 3 (July), 78:1–78:6.

PHARR, M., AND HUMPHREYS, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for GPU computing. In *Graphics Hardware 2007*, 97–106.

TURK, G. 1993. Generating random points in triangles. In *Graphics Gems*, A. Glassner, Ed. Academic Press, July, ch. 5, 24–28.

TZENG, S., AND WEI, L.-Y. 2008. Parallel white noise generation on a GPU via cryptographic hash. In *I3D '08: Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, 79–87.

WEI, L.-Y. 2008. Parallel Poisson disk sampling. *ACM Transactions on Graphics 27*, 3 (Aug.), 20:1–20:9.

WHITE, K. B., CLINE, D., AND EGBERT, P. K. 2007. Poisson disk point sets by hierarchical dart throwing. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, 129–132.